An ultra-quick primer to Fortran

Simon Scheidegger
simon.scheidegger@gmail.com
August 30th, 2018

Cowles Foundation – Yale

General information

Simon Scheidegger - simon.scheidegger@unil.ch for questions outside the class.

Fortran Intros on the World Wide Web:

wwwasdoc.web.cern.ch/wwwasdoc/f90.html

www.nsc.liu.se/~boein/f77to90/

Fortran literature:

„Fortran 95/2003“, M. Metcalf, J. Reid, M. Cohen

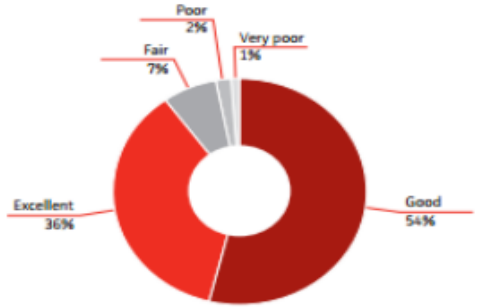
„Fortran 90/95“, Stephen J. Chapman

CSCS – Annual report 2016 – Sad, so sad

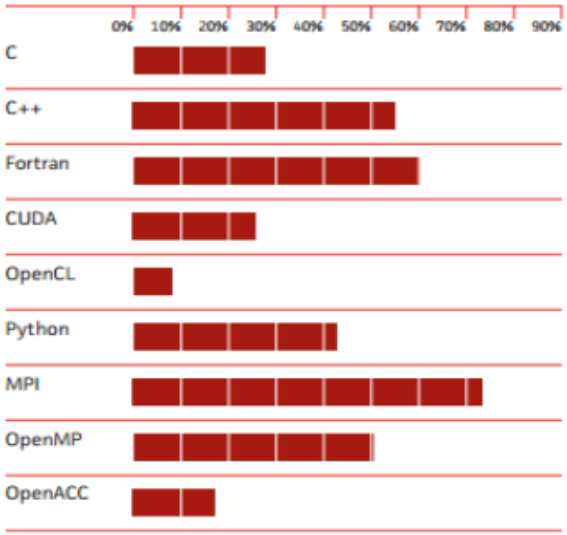
Application Development

Do you develop and maintain application codes?

How do you rate the offered range of programming tools (compilers, libraries, editors, etc.)?



Which programming languages and parallelization paradigms are you using primarily?



“Hello World” program

```
program helloworld
implicit none

write(*,*) "Hello World"

stop
end program
```

A Fortran 90 file has to be stored with the ending *.f90,

e.g. helloworld.f90

How to compile and execute a program

Compile = translate a program from a programming language to machine language.

Fortran 90 Compilers:

ifort, gfortran, pgf95, g95, pathscale, ... etc.

How do we compile e.g. helloworld.f90

```
compiler [options] program-name [-o executable-name]
```

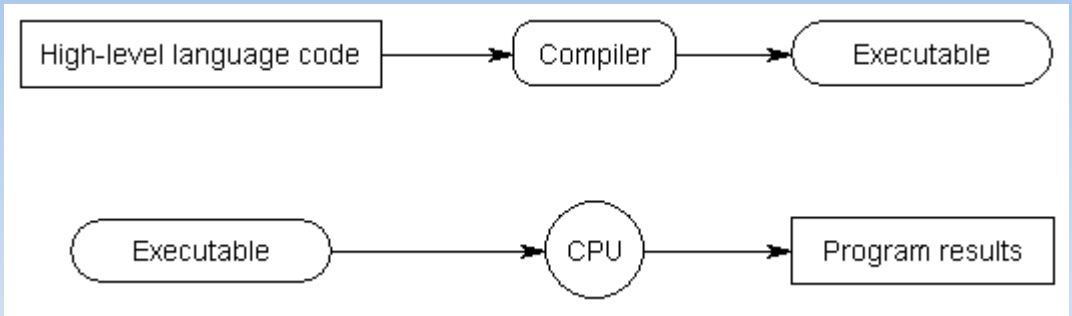
concretely:

```
gfortran helloworld.f90 -o helloworld.x
```

The compiled program can then be executed by:

```
./helloworld.x
```

The compiling process

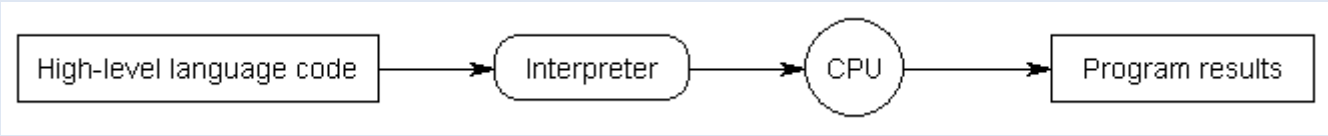


A computer's CPU is incapable of speaking Fortran (or other high-level languages).

The very limited set of instructions that a CPU natively understands is called machine code (or machine language or an instruction set).

Programs written in high level languages must be translated into a form that the CPU can understand before they can be executed. There are two primary ways this is done: **compiling and interpreting**.

Modern compilers do an excellent job of converting high-level languages into fast executables.



An interpreter is a program that directly executes your code without compiling it into machine code first. Interpreters tend to be more flexible, but are less efficient when running programs because the interpreting process needs to be done every time the program is run.

Compiling and linking

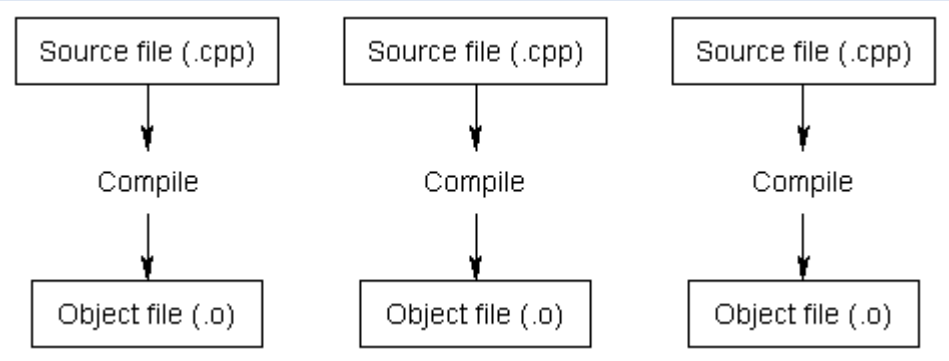
The job of the compiler is twofold:

1) To check your program and make sure it follows the rules of the programming language.

If it does not, the compiler will give you an error to help pinpoint what needs fixing.

2) To convert each file of source code into a machine language file called an object file.

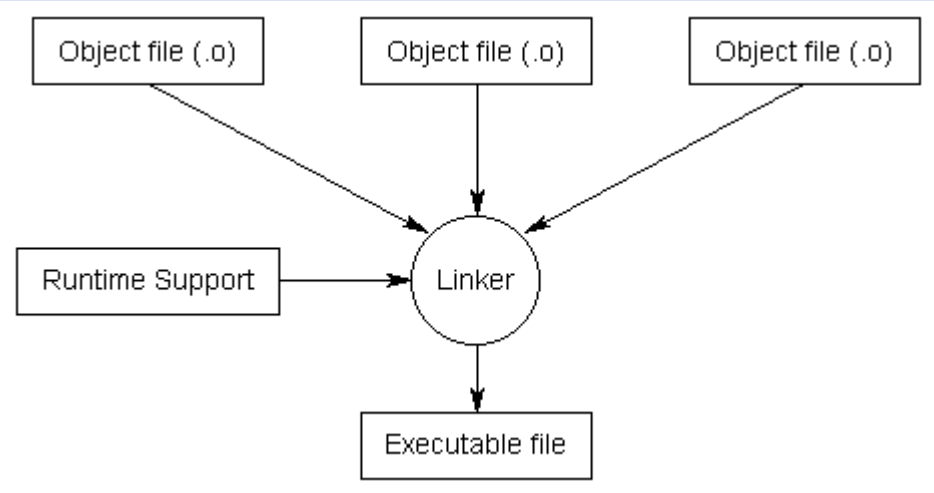
If your program had 3 .cpp files, the compiler would generate 3 object files.



Compiling and linking (2)

For complex projects, some development environments use a **makefile**, which is a file that tells the compiler which files to compile.

Linking is the process of taking all the object files generated by the compiler and combining them into a single executable program that you can run. This is done by a program called the linker.



Once the linker is finished linking all the object files (assuming all goes well), you will have an executable file.

The compile and link steps can be combined together if desired.

Compiling Code with a makefile

In case your program consists of many routines (files), compiling by hand gets very cumbersome

```
> gfortran  -o abc abc.f90 a.f90 b.f90 c.f90
```

→ **A makefile is just a set of rules to determine which pieces of a large program need to be recompiled, and issues commands to recompile them**

→ For large programs, it's usually convenient to keep each program unit in a separate file. Keeping all program units in a single file is impractical because a change to a single subroutine requires recompilation of the entire program, which can be time consuming.

→ When changes are made to some of the source files, only the updated files need to be recompiled, although all relevant files must be linked to create the new executable.

Compiling Code with a makefile (2)

Basic makefile structure: a list of rules with the following format:

target ... : prerequisites ...
<TAB> construction-commands

A **“target”** is usually the name of a file that is generated by the program (e.g, executable or object files). It can also be the name of an action to carry out, like “clean”.

A “prerequisite” is a file that is used as input to create the target.

```
# makefile : makes the ABC program

abc : a.o b.o c.o  ### by typing „make“, the makefile generates an executable denotes as „abc“

gfortran -o abc a.o b.o c.o

a.o : a.f90

        gfortran -c a.f90

b.o : b.f90

        gfortran -c b.f90

c.o : c.f90

        gfortran -c c.f90

clean :   ### by typing „make clean“, the executable, the *.mod as well as the *.o files are deleted

        rm *.mod *.o abc
```

Compiling Code with a makefile (3)

- By default, the first target listed in the file (the executable abc) is the one that will be created when the make command is issued.
- Since **abc** depends on the files **a.o, b.o** and **c.o**, all of the .o files must exist and be up-to-date. make will take care of checking for them and recreating them if necessary. Let's give it a try!
- Makefiles can include **comments** delimited by hash marks (#).
- A **backslash (\)** can be used at the end of the line to continue a command to the **next physical** line.
- The make utility **compares** the modification time of the target file with the modification times of the prerequisite files.
- Any prerequisite file that has a more recent modification time than its target file forces the target file to be recreated.

→ A lot more can be done with makefiles (beyond the scope of this lecture)

Fortran-“Alphabet“/Operators

Valid characters in Fortran:

0-9, a-z, A-Z

+ - * / ** ( ) . = , ' ! ? “ $: % & ; > <

Arithmetic operators:

** * / + -

Relational operators:

.eq. .ne. .lt. .le. .gt. .ge.

== != < <= > >=

Logical/Boolean operators:

.not. .and. .or. .eqv. .neqv.

Program structure*

```
program program-name
implicit none
! Declaration section - here, all variables
! need to be declared
double precision:: a,b,c

! Execution section - read a and b from std in
write(*,*) 'please enter two numbers'
read(*,*) a,b

! Multiplication of a and b, result displayed to
! the terminal
c = a*b
write(*,*) 'a*b = ',c

! probably invoke other, internal subroutines

! end - a program-stop can be put explicitly
! in the code
stop
end program program-name
```

*Please copy this program, compile and run it on MIDWAY.

comments are useful!

Commenting a program carefully is crucial.

In Fortran, a comment always begins with !

```
program comment
implicit none
! Here, the declaration section starts
double precision:: seconds ! If possible, the meaning of the
double precision:: hours   ! variables are commented
! Begin of main program
  read(*,*) hours ! Here, the number of hours are read in

! Spaces are treated from the compiler like a comment
  seconds = hours*3600.d0  ! conversion to seconds
  write(*,*) “Seconds: “, seconds
  stop
end program comment
```

Continuation of lines

Whenever a line ends with `&` it continues on the next one.

Increases readabiliy of the code in case of e.g. a long comment.

```
a = (b + 2*c)**3
```

```
a = (b + 2*c &
     )**3
```

```
a = (b + 2*c &
! This line is ignored by the compiler
)**3
```

Data types

There are 5 data types in Fortran 90/95

integer, real, complex, logical, character

Constants:

Integer – e.g..: 0, 1, 5, +101,- 4321, ...

Incorrect: 1,234 or 1'234 , 111.0

Real – e.g.: -1.001, 1.989e+30, 3.14159d0, 9.10E-31, ...

Incorrect: 1,02 , 99, 1.23e+4.5, 1.23-45

Complex – z.B.: (1.05, 2.45), ...

Logical .true. or .false.

Character – z.B.: 'Finance', “that's correct“, ' ', ““, 'Fraction in %', ...

Incorrect: “Hallo', 'Hello World“, 'that's wrong, this also“

Declaration of variables (1)

Always before the declaration section!!!:

```
implicit none
```

Integer numbers:

```
integer                     :: i, j, k
integer(kind=4)             :: number
integer(4)                  :: number
integer, dimension(5)       :: array1
integer, dimension(8,9):: array2
integer                     :: array1(5), array2(8,9)
integer                     :: startvalue = 100
integer, parameter          :: maxshells = 2000
```

Declaration of variablen (2)

logical data type:

```
logical:: onoff
```

real:

```
real(kind=8):: realnumber
double precision:: realnumber
```

complex:

```
complex, parameter:: complexnumber = (1,2)
```

characters:

```
character:: type
character(len=20):: name
```

Rounding

rounding of integers (towards zero):

$$\frac{1}{2}=0 \qquad \frac{3}{2}=1 \qquad \frac{5}{4}=1 \qquad \frac{9}{4}=2 \qquad -\frac{2}{3}=0 \qquad \frac{9}{2}=4$$

rounding of reals:

$$\frac{1.}{2.}=0.5 \qquad \frac{7.}{4.}=1.75 \qquad \frac{1.}{3.}=0.3333333 \qquad \frac{13.}{10.}=1.3$$

Initialization

We can fix parameters in the declaration section

```
real(8),parameter:: me = 9.10938215e-31 ! Mass of an electron
abohr = 4.d0*pi*eps0*hbar**2/me/e**2
```

It is easy to assign a single value to an entire array

```
real(8), dimension(9,11):: A
...
A = 0.d0 ! All entries of A are set to 0.
```

Attention: if a variable is used in a computation without having an assigned value, it may contain arbitrary values (depending on the compiler)

If - branching

Generally:

```
if (scalar-logical-expr) action-statement
```

```
if (scalar-logical-expr) then
  action-statement
endif [or end if]
```

e.g.:

```
if (x<0) x = 0.d0
```

```
if (x>y) then
  temp = x
  x = y
  y = temp
endif
```

If-else

examples:

```
if (x<0) then
  y = -x
else
  y = x
endif
```

```
if (x<0 .and. y<0) then
  z = -x-y
elseif (x<0 .and. y>0) then [or else if]
  z = y-x
elseif (x>0 .and. y<0) then
  z = x-y
else
  z = x+y
endif
```

Loops (1)

Loop construction:

```
! Attention: in a construct as show below,
! you MUST not forget an exit- or stop- criterion!!!
do
   ...
enddo [or end do]
```

Getting out of a do-loop with „exit“

```
do
   ...
   if ( ... ) exit
   ...
enddo
```

Loops (2)

Loop with a counter

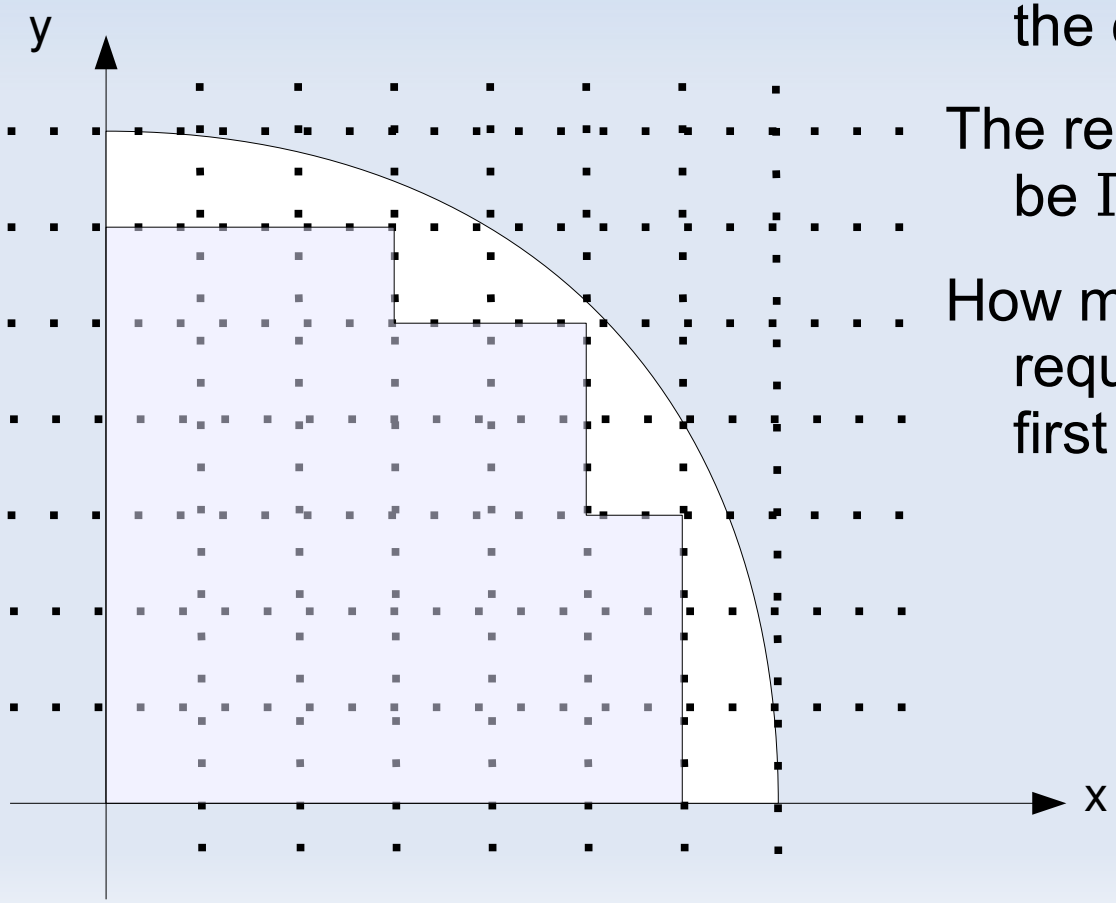
```
do count = initial-value, final-value [, step-size]
  ...
enddo
```

examples:

```
integer:: j, lower, upper
do j = lower, upper
  ...
  if (...) exit  !early exit of loop.
  if (...) stop  !brute-force stop of the program.
  if (...) cycle !the current loop index is skipped
....
enddo
```

Exercise: Approximate Π

Write a program that computes the area of the unit circle by Monte Carlo integration (example code: pi.f90).



simplification: compute a quarter of the circle.

The result should be approximately be $\Pi/4$.

How many sample points **n** are required in order to obtain the first two correct digits of Π?

Read(1)

Reading in values of variables is done by the command **read**

generally:

```
read(unit, format[,status]) [list of variables]
```

Example: the line below reads 2 values from the terminal.

```
read(*,*) Surname, Name
```

Even if there are more than 2 values provided, only the first two variables „Surname“ and „Name“ are stored.

Read(2)

read can also be used to read data from files.

```
integer:: Accountnumber, readstatus,i
real:: amount
character(len=15):: Surname
character(len=30):: Name
open(99,file='Accountdata.txt')
do
  read(99,'(a15,x,a30,2x,i10,e12.5)',IOstatus=readstatus) &
       Surname, Name, Accountnumber, amount
  if (readstatus<0) then
    write(*,*) 'There were ',i,' customer data'
    exit
  endif
  i=i+1
enddo
close(99)
```

Write

write can be used to write variables to the terminal or files.

```
write(unit,format) [list of variables to be written]
```

Write to the terminal with **write(*,[format])**:

```
write(*,*) 'Temperature= ',temp,'Density= ',density
```

In analogy to read, we can write to files:

```
integer:: i
real:: radius(nshells), temp(nshells), density(nshells)
! Here we generate an Ascii-file, and write to it
open(1,file='Dummyfile.dat',status='new')
do i=1,nshells
  write(1,'(3e17.10)') radius(i), temp(i), density(i)
enddo
```

Functions(1)

The structure of the an (external) function is similar to a main program (or subroutines). In contrast to a subroutine, a function returns a **SINGLE** value.

```
type function  function-name (arg1, arg2, ..., argn)
   implicit none
   [specification part]
   [execution part]
   [subprogram part]
end function  function-name
```

```
function  function-name (arg1, arg2, ..., argn)
   implicit none
   datatype:: function-name
   [specification part]
   [execution part]
   [subprogram part]
end function  function-name
```

Functions(2)

examples:

```
real function  f0(x)
   implicit none
   real, intent(in):: x
   f0 = cos(x)*sin(x)+ x**2
end function  function-name
```

```
function  distance (a,b)
   implicit none
   real:: distance
   real, intent(in), dimension(3) :: a,b
   distance = sqrt((a(1)-b(1))**2 + (a(2)-b(2))**2 + &
   (a(3)-b(3))**2)
end function  function-name
```

Functions(3)

Where to define a function inside a program?

Intern:

```
PROGRAM  program-name
IMPLICIT  NONE
[specification part]
[execution part]
CONTAINS
   [your functions]
END PROGRAM  program-name
```

Extern:

```
PROGRAM  program-name
IMPLICIT  NONE
[specification part]
[interface of your functions]
[execution part]
END PROGRAM  program-name

[your functions]
```

Subroutines(1)

A subroutine can, in contrast to a function, return multiple values at once.

Their form is similar to a main program or a function

```
subroutine  subroutine-name (arg1, arg2, ..., argn)
   implicit none
   [specification part]
   [execution part]
end subroutine subroutine-name
```

How can we call/access a subroutine?

```
call subroutine-name (arg1, arg2, ..., argn)
```

With **intent(in) / intent(out)** in the declaration section, we can specify which arguments are supposed to be an input or output.

Subroutines(2)

Example – an external subroutine:

```
program main-program
   implicit none
   ...
   call circle(radius, area, circumferenc)
   ...
end main-program
subroutine  circle (r,a,c)
   implicit none
   real, parameter:: pi = 3.1415926
   real, intent(in) :: r
   real, intent(out) :: a,c
   a = pi*r**2
   c = 2*r*pi
end subroutine circle
```

Intrinsic functions(1)

Numerical n:

int() - conversion into integers; rounding towards 0.

real() - conversion to a real.

dble() - conversion into double precision (real(8)).

abs() - absolute value

min() - Minimum of at least 2 values.

max() - Maximum of at least 2 values.

floor() - Rounding downwards -°

ceiling() - Rounding upwares 1 +°

...

Intrinsic functions(2)

Mathematical functions:

- Trignonometric functions: cos(), sin(), tan(), acos(), asin(), atan(), sinh(), cosh(), tanh()
- sqrt() - square route
- log() - Logarithmus naturalis
- log10() - Logarithmus with basis 10
- exp() - exponential function

Exercises

Exercise 1: Write a program that reads arbitrary values a, b, c and prints the solution to the quadratic Equation.

$$ax^2+bx+c=0$$

Exercise 2: Find the zero of the function $\cos(x)-x$ using Newton's Method.